Guard Files: Stabbing and Intersection Queries on Fat Spatial Objects

J. NIEVERGELT AND P. WIDMAYER
Informatik, ETH Zurich, CH-8092, Zurich, Switzerland

The design of spatial data structures has made great strides in recent years in response to the increasing importance of applications such as CAD that require great efficiency in spatial database technology and in computational geometry. The variety of spatial data structures and retrieval algorithms known suggests that it is difficult or impossible to design general purpose structures that perform well across the entire spectrum of objects to be stored and queries to be processed—generality comes at the cost of performance and increased algorithm complexity. Thus simple algorithms that perform efficiently on a restricted class of problems are clearly of interest.

The *guard file* is a new data structure, with its access and update algorithms, designed to answer stabbing and intersection queries on a dynamic collection of spatial objects that satisfy a shape constraint. The objects stored must be 'fat' in a technical sense, namely convex with an aspect ratio (width/length) $\geqslant f$, where f is a constant characteristic of the class, $0 < f \leqslant 1$. The collection of objects is pre-processed so every object is attached to a cell or to some vertices ('guards') of a hierarchical grid that tessellates space. The retrieval and update algorithms are simple and use standard data structures (radix tree). For restricted classes of objects, such as aligned regular polygons, efficiency may be enhanced by designing a suitable spatial grid tailored to the particular type of objects stored. We present examples, general results that relate guard grids to the fatness f of objects, and experimental results for the 2-d case. All the relevant concepts generalize readily to multi-dimensional space, though the theory becomes more complicated.

Received June 1992, revised October 1992

1. SPATIAL DATA STRUCTURES: PROBLEMS AND APPROACHES

The growing interest in spatial data structures stems from the confluence of two trends. First, the push from spatial data base technology (and its applications such as geographic information systems) for structures that efficiently support *access based on geometric criteria*, such as intersection, containment, or nearest-neighbour-properties—a variety of retrieval criteria that can be lumped under the general term *proximity queries*. Second, the pull from the thriving field of computational geometry, where powerful novel techniques raise the level of expectation about the performance to be expected. Monographs [2], dedicated conferences [1, 3], and textbooks [9] document the recognized importance of spatial data structures, and provide a survey of this expanding field.

The variety of spatial data structures and retrieval algorithms proposed suggests that it is difficult or impossible to design general purpose structures that perform well across the entire spectrum of objects to be stored and queries to be processed—generality comes at the cost of diminished performance and increased algorithm complexity. Thus simple algorithms that perform efficiently on a restricted class of problems are clearly of interest.

Spatial data structures organize space by superimposing a structure of cells of suitable shape and size. This cell structure may be static (determined *a priori*, independently of the specific objects to be stored) or dynamic (modified by every object newly inserted or deleted); it may be flat or hierarchical; cells may be disjoint or overlapping—the variety of grids investigated is extensive, since the choices listed above are not just binary, but often admit intermediate forms and compromises. As an example, consider the two extremes of cells that may overlap in arbitrary ways, and disjoint cells that partition space. In the first case, by packing every object in its own cell we gain a simple relationship between objects and cells at the expense of a random cell structure. In the second, with cells forming a regular tessellation of space, we gain a simple cell structure at the expense of a complicated relationship between objects and cells. Thus intermediate forms are of interest, where cells may overlap in restricted ways. Of these, hierarchical grids are most common: Each level of the hierarchy has its own space partition, and the relationship between these partitions is easily computed. Such hierarchical grids are the cell structures used in this paper.

All space partitioning schemes run into the same fundamental problem: When storing spatial objects, as opposed to points, we cannot easily assign an object in a natural way to a unique cell. The larger the object, the more cells it covers or intersects, the more difficult is the task of assigning an object to a representative 'home cell' that locates the object in space. By selecting a

representative point for each object, such as its centre of gravity, and storing an object in the cell of its representative point, we solve only the easy part of the problem. The harder task of retrieving an object in response to a query is not addressed at all: Although an object may cover the query cell and thus needs to be retrieved, there is no evident relation between the query cell and the object's home cell.

Since the concept of a home cell, or equivalently, a single grid point as an anchor, causes retrieval problems, many spatial data structures avoid this concept by using variations of a scheme we call 'mark inhabited space'. But each variation introduces other problems. By cutting an object into sufficiently many pieces, for example, each piece ends up in its natural, unique home cell, but retrieval and processing of the entire object may become more costly. The alternative of maintaining a pointer from every cell touched by an object to the unique description of this object makes for fast retrieval but slow updating of many pointers. In general, any scheme that replaces the concept of a single home cell or anchor by some form of 'mark inhabited space' will make it costly to move an object through space, even if the object is rigid and has a simple description. Restricted classes of objects, such as those definable by some small, fixed set of parameters, yield to simpler solutions, such as transforming an object into a point in parameter space [6]. Many schemes, as well as combinations and hybrids, have been described in the literature. We refer to our surveys [7, 12] for an overview and classification.

This paper presents the *guard file*, a data structure designed to answer stabbing and intersection queries on a dynamic collection of spatial objects that satisfy a shape constraint. The objects stored must be 'fat' in a technical sense, namely convex with an aspect ratio (width/length) $\geq f$, where f is a constant characteristic of the class, $0 < f \leq 1$. 'Fatness' makes these objects sufficiently 'spherical' that we can salvage the idea of assigning each one to a single home cell, or anchoring it at a grid point, while retaining an efficiently computed relationship between any query cell and the object's home cell or anchors. The collection of objects is preprocessed so every object is attached to a cell or to a small, constant number of vertices ('guards') of a hierarchical grid that tessellates space. The retrieval and update algorithms are simple and use standard data structures (radix tree). For restricted classes of objects, such as aligned regular polygons, efficiency may be further enhanced by designing a suitable spatial grid tailored to the particular type of objects stored. We present examples, general results that relate guard grids to the fatness f of objects, and experimental results for the 2-d case. All the relevant concepts generalize readily to multi-dimensional space, though the theory becomes more complicated.

Are restricted classes of objects defined by properties such as convex, 'fat', or aligned (i.e. with sides parallel to a given set of directions) relevant in practice? Although such restricted objects do occur in a few applications (aligned rectangles used in integrated-circuit design are a prominent example), the vast majority of applications (e.g. geographic information systems, computer-aided design of machinery or of buildings) rely on objects of a great diversity of shapes. What then, one may ask, is the role of data structures that can handle only a few types of spatial objects?

The answer is that, the more complicated and diverse the objects to be processed, the more it pays to pack them into simple containers: A comb-shaped object might be packed into a rectangular bounding box, a wheel with spokes into a circle, any object might be data-compressed into its convex hull. The point of packing irregularly-shaped objects into simple containers is to process queries (such as intersection queries) in two steps: First, against the collection of containers, thus eliminating most containers; second, against the objects in only those containers that survived the first filter. Since containers are chosen to have simple shapes, this two-step process greatly enhances efficiency. Of course there are important spatial objects, such as roads or rivers, that resist being squeezed efficiently into simple containers. Nevertheless, the technique of packing complicated objects into simple containers is widely-used and motivates the search for data structures that handle only restricted classes of objects, but do so very efficiently.

2. STABBING AND INTERSECTION QUERIES: CONCEPTS, TERMINOLOGY, NOTATION

We consider a class of problems of which the following is a typical simple example:

Given a collection D of circles embedded in the plane, design data structures and corresponding retrieval and update algorithms to efficiently answer stabbing queries of the type: For an arbitrary query point q in the plane, retrieve all circles $d \in D$ that overlap q, i.e. $q \in d$.

In general, we consider intersection query problems of the following type.

Given:

A (usually ∞) class S of spatial objects embedded in k-dimensional Euclidean space R^k,
A finite subset $D \subset S$ of data, i.e. objects $d \in D$ to be stored; $|D| = n$,
A (usually ∞) class Q of query regions $q \subset R^k$,

Answer intersection queries:

For an arbitrary query $q \in Q$, list all objects $d \in D$ with $d \cap q \neq \{\}$.

In the example above, S is the set of all circles embedded in R^2, D a finite subset of n circles, and Q the set of points in R^2 (the special case of point queries is often called 'stabbing queries').

The solution sought consists of *data structures* and

corresponding *retrieval and update algorithms* to answer stabbing and intersection queries efficiently. These will naturally depend on the object class S and query class Q. The solution presented, the *Guard File*, works well for *homogeneous* classes S consisting of *simple* objects that meet a certain *shape constraint*. Let us clarify these concepts:

- 'Simple' and 'homogeneous' are intuitive concepts. For guard algorithms to be practical, objects must have simple and similar descriptions. The theory can be extended to inhomogeneous classes of complicated objects, but at the cost of correspondingly more complicated algorithms.
- The shape constraint states that the objects must be '*fat*'. Fatness can be defined in different ways, but always implies two conditions: The objects are *convex* with an *aspect ratio* (width/length) $\geq f$, where f is a constant characteristic of the class, $0 < f \leq 1$. Definitions may differ in how width and length of a convex object are defined. The exact definition of fatness naturally influences the exact results one obtains, but the latter are always of the same form: They relate object fatness to the data structure needed to manage the objects, and to the number of probes needed to answer queries.

The solution sought also assumes that the data set D is dynamic, subject to insertions and deletions of individual data objects d—the usual case in applications. It follows that the data structure and algorithms must be designed based only on information about S, Q, and the fatness parameter f, independently of D. Two other cases where D is assumed static occur more rarely in practice:

1. If D can be inspected before the data structures are designed (as in the case of perfect hashing, for example), it may be possible to design algorithms of superior performance that are tailored to the specific data set D; the ideas behind the guard file may well apply to this case.
2. If we build the data structures according to the general scheme to be described, based merely on knowledge of S and Q, then the assumption of no insertions into D or deletions from D does not appear to simplify the algorithms, as updating the radix tree used by a guard algorithm is a rather simple operation anyway.

Guard algorithms are designed to process data efficiently off disk, in that objects that are close are likely to be stored in the same data bucket. Under reasonable assumptions about the data distribution, most of the objects that might cover a query point q (i.e., either cover q or come so close that they must be inspected) are stored in just a few data buckets that must be read to process the query.

3. EXAMPLE: STABBING QUERIES FOR CIRCLES AND ALIGNED REGULAR HEXAGONS

The concept of 'guard file' is general enough to allow different realizations adapted to distinct problems. The key ideas are best explained by an example, rather than a theory formulated in terms sufficiently general to encompass all the cases we wish to include. We will continue with the introductory example of storing a collection D of circles embedded in the plane so as to efficiently answer stabbing queries. But the explanation turns out to be simpler if we begin by approximating a circle by an aligned regular hexagon, i.e. by a hexagon with sides parallel to the boundaries of a given grid of triangular cells. Circles are harder to deal with than hexagons because of a technical difficulty soon to be encountered. We develop these ideas in three steps:

1. A grid of fixed size illustrates the concepts of cells and guards, and their functions.
2. A hierarchical grid is used to illustrate a complete guard algorithm.
3. Various solutions to the difficulty posed by circles illustrate the flexibility inherent in guard algorithms.

3.1. A cell and its guards

Given an equilateral triangle T_0 in the plane, let S_6 be the set of regular hexagons aligned with T_0, i.e. every side of every hexagon is parallel to some side of T_0. Let the equilateral triangle T_1, shown in the figure below, be the 'centre one-fourth' of T_0, and assume initially that the query set Q is the set of points interior to T_1. Figure 1 shows 5 hexagons d_0, d_1, d_2, d_3 and d_4 from the data set $D_6 \subseteq S_6$, and a query point q. We call T_1 the 'cell of q', a concept to be refined later when considering hierarchical grids.

None of the hexagons happens to cover the query point q, but d_1 and d_2 come close, in the sense of intersecting $T_1 = cell(q)$. Our example involves point queries for simplicity's sake, but the relevant concepts generalize directly to region queries q, where $cells(q)$ denotes the set of cells intersected by q. We call an object $d \in D$ a '*candidate*' w.r.t. a (point- or region-) query q if it intersects $cell[s](q)$. C or $C(q)$ denotes the cell or cells intersected by a query q.

The guard file is designed to quickly recognize candidate objects and filter out non-candidates. The decision whether a candidate actually intersects the query q is left to a routine specific to the object class S and the query class Q. For the purpose of this paper, any query q is identified with the cell[s] $C(q)$ intersected by q, and processing ends when the set of candidates has been by identified.

In Figure 1, neither d_0 nor d_3 nor d_4 is a candidate, but we can still make a useful distinction between them. We call d_3 and d_4 '*suspect*' because if they were slightly larger, as are d_1 and d_2, they might intersect $cell(q)$, and thus turn into candidates, without any 'topological'

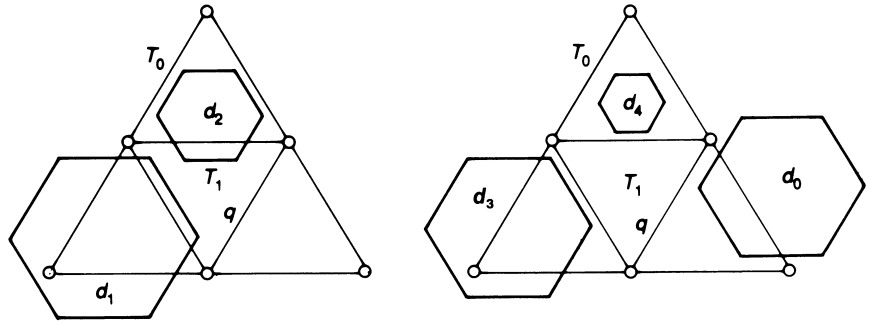

FIGURE 1. Candidate objects d_1 and d_2, suspects d_3 and d_4, and non-suspect d_0.

change in the sense of covering any new guards of *cell*(q), i.e. the vertices of T_0 and T_1. On the other hand, d_0 is not suspect: If we enlarge d_0 to intersect *cell*(q) it will bump into a vertex, i.e. a guard. The guard file works on the principle that *only candidates and suspects need be examined*: It attacks the problem of efficiently selecting candidates and suspects from the typically much larger set D, and filtering out non-suspects without wasting any time on them. The intuitive idea behind the guards approach is to distinguish objects that are guaranteed to be small from those that are potentially large, and to treat the two types separately. The reason is that we wish to locate, or anchor, a data object in that region of space where its centre lies, and from this point of view small objects are much simpler to locate than large objects:

- For a small object such as d_2 to be a candidate, i.e. intersect *cell*(q), it must have its centre in *cell*(q) or in its immediate vicinity.
- A large object, on the other hand, may have its centre arbitarily far away. In order to avoid searching the entire space, we aim to ensure that any large object that intersects *cell*(q), such as d_1, 'gets caught' by 'guards' posted around $C(q)$; in other words, any large object that escapes q's guards cannot possibly cover q. If this condition is met, we only have to inspect the objects caught by q's guards.

Let the vertices of T_0 and those of T_1 be the 6 guards of $T_1 = cell(q)$, and partition the hexagons in D into two classes:

- the guarded hexagons are those that overlap at least one guard (e.g. d_1, d_3 in Figure 1),
- the unguarded hexagons overlap no guards (e.g. d_0, d_2, d_4).

Observe the following elementary geometric fact, the '*guard lemma*', stated in two equivalent ways:

1. If hexagon d intersects T_1, then d is either guarded, or is an unguarded hexagon with its centre in T_0.
2. An unguarded hexagon d with its centre outside T_0 cannot intersect T_1.

This geometric fact serves as a filter for excluding parts of the data set D from inspection. In order to answer the stabbing query for any q in $T_1 = cell(q)$, we must inspect all guarded hexagons, but only those unguarded hexagons whose centre is in one of 4 cells whose union is the circumscribed triangle T_0: namely, *cell*(q) and its 3 adjacent cells.

3.2. Hierarchical grids

The grid used above, consisting of triangles T_0 and T_1, shows how the guards of a cell serve to exclude from inspection data objects that cannot possibly be candidates or suspects. In designing an algorithm for rapid retrieval of the candidate hexagons for any query q in some query set Q, we use this idea at varying levels of granularity and introduce space tessellations consisting of an arbitrary number of cells, not just 4 as in the example above. With a slight change of assumptions, consider the triangle T_0 to be our universe in the sense that S is the set of all hexagons with centre in T_0, and Q is the set of interior points of T_0. We introduce a hierarchical triangular tessellation of T_0, as illustrated in the figure below.

T_0 is the only cell at level 0, or 0-cell; its 3 vertices are called 'level 0 guards'. We partition T_0 into $4 = 4^1$ triangles T_{00}, T_{01}, T_{02}, T_{03} shown at left, called cells at level 1, or 1-cells; their vertices introduce 3 new level 1 guards. Proceeding recursively, we obtain 4^2 new cells at level 2, called 2-cells, which introduce 9 level 2 guards. This recursive partition defines a radix-4 tree with T_0 at the root, 4 children T_{00}, T_{01}, T_{02}, T_{03} of the first generation, and so on, down to some depth or height H which determines the desired resolution of partitioning the universe into smallest cells. For an arbitrary point

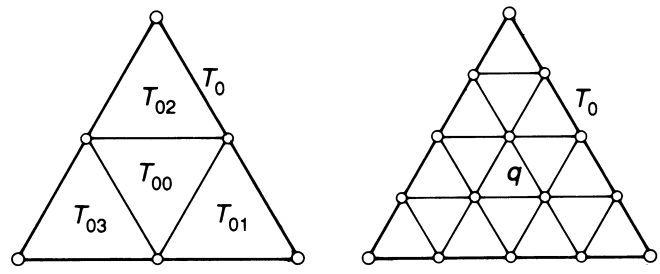

FIGURE 2. The top levels of a hierarchical triangular tessellation of T_0.

q, let $cell_h(q)$ denote the cell at level h, or h-cell, that contains q. In particular, $cell_H(q)$ is the finest cell containing q, and $cell_0(q) = T_0$ is the coarsest cell. Figure 3 shows a typical node of the radix tree and its 4 children; the node corresponding to the central cell introduces 3 new guard points.

In considering how to organize the data set D for rapid retrieval of the candidate and suspect hexagons for any query q in T_0, we choose a different scheme for guarded and unguarded hexagons.

1. Each unguarded hexagon is uniquely associated with the finest cell of its centre point p, $cell_H(p)$, and every H-cell has a data bucket to store all its unguarded hexagons. Given a query q, say by its coordinates (x, y), a simple and rapid computation yields the unique identifier of $cell_H(q)$, from where a pointer leads directly to its bucket.
2. Each guarded hexagon is associated with some guard[s]. Whereas an unguarded hexagon is naturally associated with a unique cell, a guarded hexagon may cover any number of guards and thus is not readily associated with any one guard. If every guard points to every hexagon that covers it, such unbounded multiple references introduce redundancy that leads to inefficient update algorithms. A simple rule bounds the number of multiple references by a small constant (3 in this example): Among all the guards covered by a hexagon d, let (all and only) the highest-level guards point to d.

With this convention, a stabbing query for an arbitrary point q is answered as follows:

1. Retrieve and examine all unguarded hexagons d with centre in $cell_H(q)$ and in the $\leqslant 3$ adjacent H-cells.
2. Retrieve and examine all guarded hexagons d along the path from root to $cell_H(q)$. The 'path' consists of exactly one cell at each level h, with the guards at its 3 vertices.

The discussion so far ignores a technical detail: How to define the meaning of 'a point p lies in cell c' in such a way that every point of T_0 belongs to a unique cell at given level h, a problem that arises only for points p on cell boundaries. Everything can be made to work even if we fail to disambiguate the cell membership of boundary points: For example, by storing an unguarded hexagon whose centre lies on a boundary in both adjacent cells, and by treating a query q on a cell boundary (but not on a guard point) as if q belonged to both adjacent cells. A query q on a guard point is easy to handle, since only guarded circles can cover it, namely those on the path from the root to the guard at q.

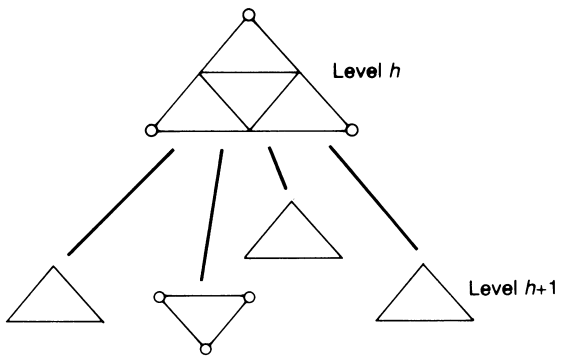

FIGURE 3. Hierarchical triangular tessellation represented as a radix 4 tree.

But it is both more efficient and more elegant to disambiguate the cell membership of a boundary point. Among many ways of doing so we suggest the following scheme. Notice that at every level h, 'upright triangles' (with a vertex on top, oriented the same way as T_0) alternate with 'triangles that stand on their head', like T_1. We define an upright triangle to be a closed point set, i.e. to contain the points on its boundary; whereas a triangle that stands on its head is open and contains only the points in its interior. This takes care of all points except guard points, which are vertices of up to three upright triangles. It is unnecessary to disambiguate the cell-membership of vertices, as any hexagon with its centre on a vertex is guarded and thus gets stored with some guards, not with any cell.

3.3. How to catch circles in a triangular grid

This completes the outline of a guard algorithm for answering stabbing queries on aligned regular hexagons embedded in the plane. In considering the similar example where S is the set of circles we run into a tricky problem: Almost the entire argument used for hexagons carries over, except for one crucial detail. The key 'guard lemma', rephrased in terms of circles rather than hexagons, reads: "Observe the following elementary geometric fact stated in two equivalent ways:

1. If circle d intersects T_1, then d is either a guarded circle or an unguarded circle with its centre in T_0.
2. An unguarded circle d with its centre outside T_0 cannot intersect T_1."

But unfortunately in this form it is wrong! There are unguarded circles, such as d_1 in Figure 4, with centre (slightly) outside T_0 that intersect $cell(q)$—just barely!

From the point of view of developing a theory of guard algorithms, this example of 'it almost works, but not quite' is not a big hurdle, as we can readily think of ways to catch the few wayward unguarded circles such as d_1 that sneak into $cell(q)$ from a distance:

1. A brute-force fix—general, simple, costly. We recognize that it is insufficient to inspect the unguarded circles in $cell(q)$ and its 3 adjacent cells. Rather, we must widen the protective rim around q to include 6 cells at distance 2 from $cell(q)$, labelled 2 in the figure below, and further described in the section where we analyze triangular grids. Anticipating the terminology of section 4, this brute-force solution protects $cell(q)$ by a cell-disk of radius $r_C = 2$ with $c(2) = 10$ cells to

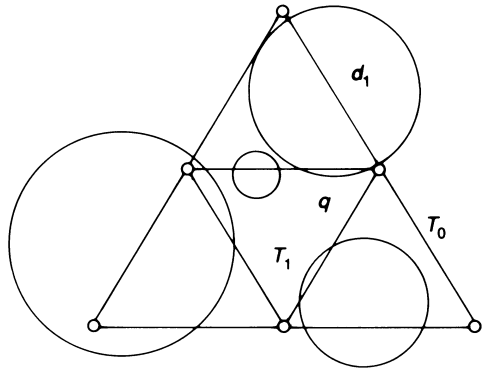

FIGURE 4. The algorithm developed for hexagons fails for circles: unguarded circle d_1 with centre outside T_0 nevertheless intersects T_1.

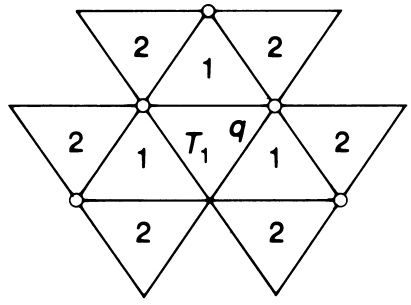

FIGURE 5. $T_1 = cell(q)$ with a protective cell-disk of radius 2.

be inspected, and a guard-disk of radius $r_G = 1$ with $g(1) = 6$ guards to be inspected at each level of the hierarchy.

2. 'Haloing'. We recognize that a 'wayward' unguarded circle such as d_1 can miss a guard by at most some small quantity ε that depends on the cell size. By redefining a circle $d \in D$, of radius r, to be 'guarded' if its slightly expanded ('haloed') circle d' of radius $r + \varepsilon$ covers a guard, the wayward unguarded circles become guarded by definition. The protective rim around q required to catch unguarded circles remains at a total of four cells, but we must inspect a (slightly) larger number of guarded circles that are not part of the answer.
3. A speculative issue, raised only as an illustration of the flexibility inherent in guard algorithms. Plausibly, but without justification, we have assumed that the cell T_1 is guarded by the 6 vertices of T_0 and T_1. But the main purpose of guards is to make the guard lemma true: 'An unguarded circle d with its centre outside T_0 cannot intersect T_1', and for this there might be better ways of posting guards than insisting on cell vertices! E.g., by retaining as guards the 3 vertices of T_1, but moving the other 3 guards slightly inwards from the vertices of T_0, the newly posted 6 guards will protect cell T_1 perfectly. The trouble with this particular solution is apparent when considering hierarchical grids, where we must protect not only T_1, but also its sibling cells. Then we notice that a single guard at a vertex of T_1 protects up to 6 cells reasonably well, whereas a guard 'shifted inwards' to better protect T_1 does a poor job of protecting T_1's neighbours and might have to be supplemented with additional guards.

So far we have introduced, by example, most of the concepts and techniques needed to design a guard file and its access algorithms. Notice that an algorithm designer retains much freedom in designing grids, guards, and protective rims tailored to exploit geometric features of the class S of objects to be stored. Thus guard algorithms constitute a rather general approach to object retrieval. They rely on the presence of only one property that objects must possess: They must be 'fat', so as to guarantee that an object whose centre of gravity is far away cannot sneak into a cell unobserved by the cell's guards.

4. HIERARCHICAL CELL AND GUARD GRIDS

A variety of hierarchical grids may serve for posting guards, and some grids are more efficient than others for a specific class of objects to be stored. Next to the triangular grid of Section 3, we consider two other useful hierarchical grids: the well-known square grid used in quad trees, and the hexagonal grid.

The structure of the hierarchical hexagonal grid differs from the triangular and the square grid in that a cell at level h is partitioned not only into whole cells at level $h + 1$, but rather into a combination of cells and parts of cells. Since a cell at level $h + 1$ may be shared by 3 cells at level h (may have 3 parents), the 'hierarchy' is not a tree, but a directed acyclic graph (dag). This causes no problems, since guard algorithms impose only two important requirements on a hierarchical grid:

1. a regular structure that permits efficient address computation and thus a fast computation of the path to be followed from the root of the dag to a leaf, and
2. at each level h, the h-cells tessellate the entire space.

As Figure 6 shows, a hierarchical grid defines a sequence of refined regular tessellations of space into cells, one tessellation per level h. When we emphasize cells, we speak of *cell grid*. For a given cell grid there may be several reasonable ways of posting guards, as

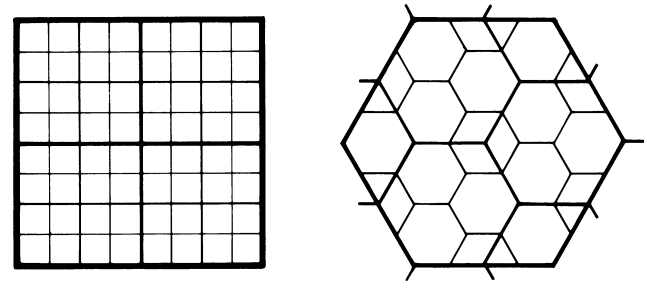

FIGURE 6. Two other hierarchical grids: Quad-tree and hexagonal grid.

we discussed in Section 3.3, so we also need to define a corresponding *guard grid*. In this paper we follow the convention that the guards are precisely the vertices of cells, and we may use two near-equivalent ways of assigning a level to these guards:

1. a vertex of an h-cell has a guard at level h; this implies that, since one and the same point in the plane may be a vertex of cells of many different levels, it may also have multiple guards of different levels; or
2. each vertex has exactly 1 guard, whose level is the highest among all the cells of which it is a vertex.

In devising an algorithm to answer a stabbing query for q, we set up different rules to catch the unguarded objects and the guarded objects. For the unguarded objects, it suffices to inspect the leaf $cell_H(q)$ and a sufficiently wide protective rim of leaf cells in its vicinity. For guarded objects, it is sufficient to inspect a carefully chosen subset of all the guards, the 'active guards'; at each level h, the active guards are in the h-vicinity of $cell_h(q)$. In order to characterize useful neighbourhoods of an h-cell c, we define two types of *distances* among cells *within a given level h* of the hierarchy: *e-dist* is based on edge-adjacency, *v-dist* on vertex-adjacency.

Definition. Cell distances. Every cell is at distance 0 from itself. An h-cell c'' is at distance i from h-cell c if

1. c'' shares an *edge* (in the case of *e-dist*) or a *vertex* (in the case of *v-dist*) with an h-cell c' at distance $i-1$ from c, and
2. i is the smallest such integer.

For both *e-dist* and *v-dist*, Figure 7 shows rings of equidistant cells in triangular, square, and hexagonal grids.

The following definitions all come in two versions, depending on whether we use the *e-dist* or the *v-dist* metric. As a rule of thumb, *e-dist* defines useful neighbourhoods for aligned objects, whose edges are parallel to grid edges; whereas *v-dist* is more useful for arbitrary objects.

Definition. Cell-disk, guard-disk. The cell-disk of radius $r \geqslant 0$ around h-cell c consists of all h-cells at distance $\leqslant r$ from c. The guard-disk of radius $r \geqslant 0$ consists of all guards in c's cell-disk of radius r.

Notation:

- r_C = radius of a cell-disk.
- r_G = radius of a guard-disk.
- c or $c(r) = \#$(cells in a cell-disk of radius r), usually the number of cells to be inspected.
- g or $g(r) = \#$(guards in a guard-disk of radius r), usually the number of guards to be inspected.

Figure 8 shows the values of $c(1)$ and $g(1)$ for the grids and metrics we consider.

5. DEFINITION AND PROPERTIES OF FAT OBJECTS

The property of objects being fat works to our benefit for both guarded and unguarded objects: Intuitively, the fatter an object is, the fewer guards and cells are required to catch it. Whereas a long, thin object with its centre far away can sneak through the grid of guards and intersect $cell_h(q)$, a fat object that intersects $cell_h(q)$ and has its centre far away must cover (and get caught by) some high-level guard of $cell_h(q)$.

We capture the intuitive concept of 'fatness' by insisting on two conditions necessary to make guard algorithms work: The objects are *convex* with an aspect ratio $(width/length) \geqslant f$, where f is a constant character-

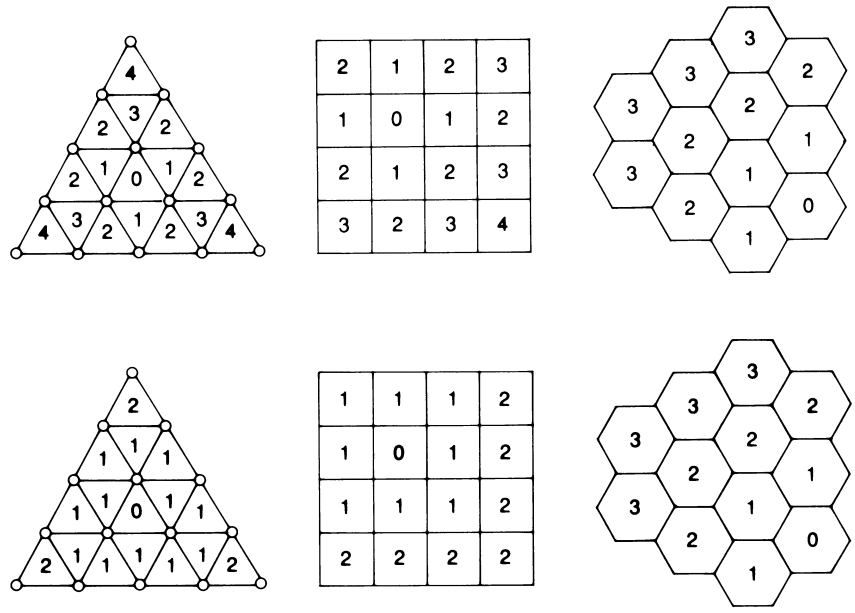

FIGURE 7. *e*-distances (top) and *v*-distances (bottom) in various grids. No difference for hexagonal grid.

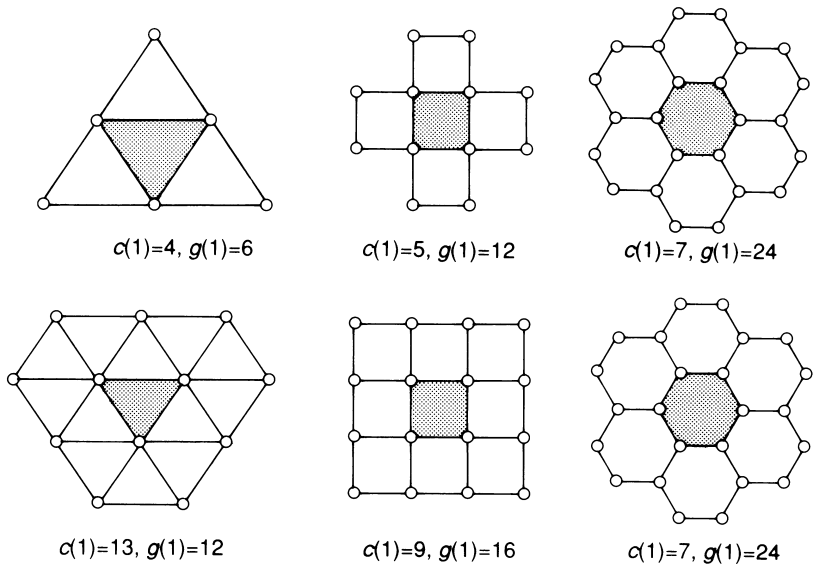

FIGURE 8. Number of cells and guards in protective disks of radius 1 in various grids. For e-distances (top) and v-distances (bottom). No difference for hexagonal grid.

istic of the class, and $0 < f \leqslant 1$. There remains the issue of defining 'width' and 'length'. For objects aligned with the grid (e.g. the hexagons of Section 3), it is natural to restrict the directions in which width and length are measured to a few directions characteristic of the grid. Example: To store aligned rectangles in a square grid, we would develop an algorithm that measures width and length along the horizontal and vertical axes only, and expect it to be more efficient than one based on a definition that allows width and length to be measured in any direction. But in this Section we consider convex objects of arbitrary orientation, for which *rotation-invariant definitions of fatness* are appropriate. The following definitions may be of general interest: One is based on the aspect ratio of enclosing rectangles, another on fractional area [8], and the third on distances within the object.

Definition. The r-fatness $f_r(d)$ of a convex object d is the minimum aspect ratio width/length of a rectangle of arbitrary orientation that tightly encloses d.

Examples. $f_r(\text{circle}) = f_r(\text{square}) = 1$; $f_r(\text{equilateral triangle}) = \sqrt{3}/2 \approx .87$.

Definition. The a-fatness $f_a(d)$ of a convex object d is the ratio $A(d)/A(c)$, where $A(d)$ is the area covered by d and $A(c)$ is the area of the minimal enclosing disk c that covers d.

Examples. $f_a(\text{circle}) = 1$; $f_a(\text{square}) = 2/\pi \approx .64$; $f_r(\text{equilateral triangle}) = 3\sqrt{3}/(4\pi) \approx .41$.

We have chosen to study the consequences of the technically most convenient concept, to be called cut-fatness $f_c(d)$, presented in the following definitions and illustrated in Figure 9.

Definition. A cut s of a convex object d at a straight line L that intersects the interior of d is the intersection of d with L.

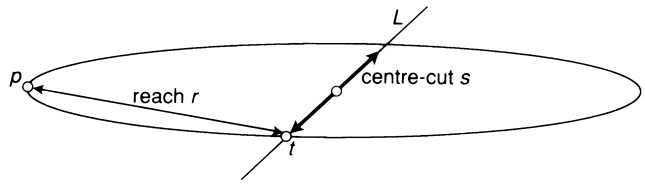

FIGURE 9. Centre-cut and reach of an object d w.r.t. a given line L.

A cut is a line segment that cuts the object into two pieces. s denotes both the line segment and its length.

Definition. A centre-cut is a cut through the centre of gravity of object d.

We are mainly interested in short centre-cuts, which intuitively measure the 'width' of d.

Definition. The reach r of a convex object d w.r.t. a centre-cut s is the maximum Euclidean distance of any point p of d to the cut s: $r = \max_{p \in d} \min_{t \in s} dist(p, t)$.

Intuitively, the reach of an object is half its 'length', or the extent to which it can penetrate into a cell if its centre-cut s is held back at the cell boundary.

Definition. The cut-fatness $f_c(d) = \min_L s/2r$ of a convex object d is half the minimum ratio of the length of a centre-cut s and the reach r w.r.t. s.

Figure 10 shows that cut-fatness coincides with the intuitive ratio 'width'/'length' for symmetric objects.

Examples. $f_c(\text{circle}) = 1$; $f_c(\text{equilateral triangle}) = 1/\sqrt{3} \approx .58$; $f_c(\text{regular } p\text{-gon}) < f_c(\text{regular } (p+1)\text{-gon})$.

For any notion of fatness considered we define:

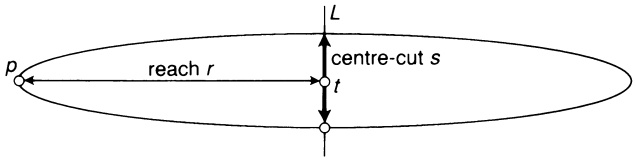

FIGURE 10. A case where $f_c(d) = s/2r$ = 'width'/'length'.

Definition. The fatness $f(S)$ of a set S of objects is the smallest fatness of any object in S.

The fatness $f(S)$ of a class of objects is used to determine the size of the protective rim of cells and guards needed to answer queries. The 'cut-lemma' illustrated in Figure 11 establishes a quantitative relationship by bounding the extent to which an object d can penetrate into a cell in terms of d's fatness $f_c(d)$.

Cut Lemma. Given a convex object d, a cut s of d, and a point p of d, such that p and d's centre of gravity are on opposite sides of s. Then

$$f_c(d) \leqslant s/2dist(p, s) \quad \text{or equivalently} \quad 2 \cdot dist(p, s) \leqslant s/f_c(d)$$

Proof. Let cs be the centre-cut of d parallel to s. By convexity of d, cs is contained in the line segment $t = (t', t'')$ whose endpoints t', t'' are the intersections of the extended line of cs with the two lines L', L'' through p and the endpoints s', s'' of s. By the definition of fatness and the similarity of the two triangles $\Delta(p, t', t'')$ and $\Delta(p, s', s'')$ we have:

$$2 \cdot f_c(d) \leqslant cs/dist(p, cs) \leqslant t/dist(p, cs) \leqslant s/dist(p, s)$$

The inequality above holds both for the case shown in Figure 11, where the distance from p to s involves an endpoint s'' of s, as well as when $dist(p, s)$ is measured by a segment orthogonal to s. Q.E.D.

6. FATNESS REQUIRED FOR VARIOUS GRIDS

The cut-lemma readily yields results of the type: For a given hierarchical grid and a class S of objects of fatness $f_c(d)$, a protective cell- and guard-disk of radius r serves to answer stabbing queries, and no smaller disk will do in general. We illustrate this claim with typical results together with the geometric arguments used to obtain them. The grid metric used in this section is *v-dist* based on vertex adjacency.

The following results are best-possible in the sense that, for a given grid and protective disk, the stated fatness bound cannot be improved. But there remains plenty of room to change the rules of the game and thereby hope to improve the efficiency of guarding objects, measured in the number of cells and guards to be inspected. For example, by using the haloing technique described in Section 3.3, or by studying other cell configurations and guard placements than the cell- and guard-disks of radius r used below.

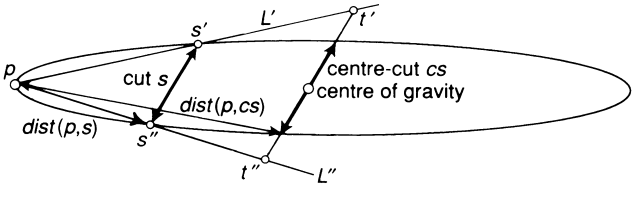

FIGURE 11. Cut-lemma illustrated for the case where $dist(p, s)$ is measured at an endpoint s'' of s.

6.1. Square grid

THEOREM. (a) A protective disk of radius 1 guards objects of any class S of fatness $f_c(S) \geqslant 1/2$. (b) In general, objects leaner than 1/2 (i.e. of fatness $f_c(d) < 1/2$) cannot be thus guarded.

Proof:

(a) As shown in Figure 12, assume that the centre of gravity g of an object d is outside the 3×3 cells, and one of the 4×4 guards lies in d, yet d penetrates into the centre cell. Let p be a point of d in $cell_H(q)$. Since d must pass between two adjacent guards, there exists a cut $s < 1$ (the grid unit). The distance from p to s is at least 1. By the cut lemma, we get $f_c(d) < 1/2$.

(b) Figure 12 also shows a class of objects of fatness $f_c(d) < 1/2$, but arbitrarily close to 1/2, that cannot be guarded by a protective disk of radius 1. Q.E.D.

Object classes of fatness $f_c \geqslant 1/2$ that can be guarded by a protective disk of radius 1 include circles and regular p-gons, i.e. equilateral triangles, squares, etc.

By enlarging the protective rim to a disk of radius 2 and increasing the cost from $c(1) = 9$, $g(1) = 16$ to $c(2) = 16$, $g(2) = 25$ we halve the required fatness:

THEOREM. A protective disk of radius 2 guards objects of any class S of fatness $f_c(S) \geqslant 1/4$. Objects leaner than 1/4 cannot be thus guarded.

Proof. Same as above, except that 'The distance from p to s is at least 2', as Figure 13 shows.

6.2. Triangular grid

THEOREM. A protective disk of radius 1 guards objects of any class S of fatness $f_c(S) \geqslant 1/\sqrt{3} \approx .58$. Leaner objects cannot be thus guarded.

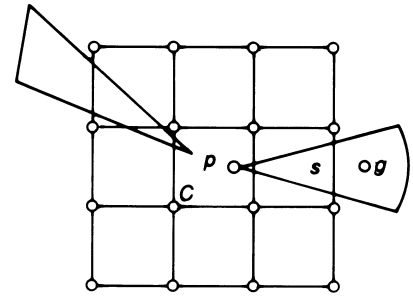

FIGURE 12. Cell C penetrated at p by an unguarded object with centre of gravity g outside the protective disk.

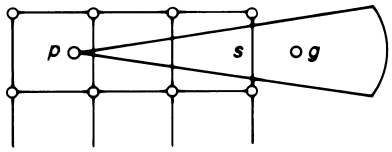

FIGURE 13. A protective disk of radius 2 guards objects of fatness $f_c(S) \geq 1/4$.

Proof. Similar to the square grid, but consider Figure 14a. The distance between a point p in the cell C to be guarded, and a cut $s < 1$ (the grid unit) along the perimeter is $\geq(\sqrt{3})/2$. Q.E.D.

Although a protective disk of radius 1 in a triangular grid protects against fewer objects than in a square grid, it suffices to guard circles and regular p-gons. Let us compare the triangular and the square grid for storing objects of a given class S. Depending on the fatness $f_c(S)$, one or the other grid is more efficient, in terms of the number of cells at level H and of guards at each level that need to be inspected in a query. For $f_c(S) \geq 1/\sqrt{3}$, the triangular grid needs only 12 guards per level as against 16 for the square grid, whereas 13 as against 9 cells are needed on level H. For $1/\sqrt{3} > f_c(S) \geq 1/2$, the square grid is better, since a protective ring of radius 1 is insufficient in a triangular grid.

6.3. Hexagonal grid

THEOREM. A protective disk of radius 1 guards objects of any class S of fatness $f_c(S) \geq 1/2$. Objects leaner than 1/2 cannot be thus guarded.

Proof. Similar to the square grid, but consider Figure 14b. The distance between a point p in the cell C to be guarded, and a cut $s < 1$ (the grid unit) along the perimeter is ≥ 1. Q.E.D.

Compared with the square grid, the hexagonal grid uses fewer cells (7 instead of 9), but 50% more guards (24 instead of 16). The comparison of grids in terms of guard and cell efficiency immediately suggests the open problem of identifying optimal grids for various classes of objects.

7. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We have experimented with two implementations of guard files [11, 5]. Peter Skrotzky implemented guard algorithms for several grids and object classes to run on the XYZ GeoBench [10], a programming environment used to develop a program library for geometric computation. He emphasized tutorial algorithm animation to explain object insertion and retrieval. Nguyen Viet Hai wrote a stand-alone C program for square grids, with aligned rectangles both as data objects and range queries. He emphasized performance measurements using both random and actual geographic data. His experiments show how various performance criteria can be improved by adapting the guard file to known statistical parameters of the data. An encouraging outcome of these experiments is that the very first guard file implementation performed about as well as a highly optimized R-file program [4].

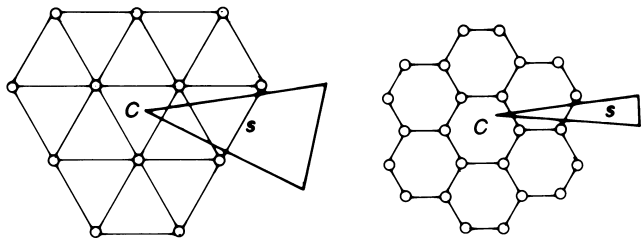

FIGURE 14. Protective disks of radius 1 for (a) triangular and (b) hexagonal grids, with critical objects.

Acknowledgement

Thanks to Peter Skrotzky and Nguyen Viet Hai for implementing test versions of guard files, to Peter Schorn and Oliver Günther for advice, and to the Swiss National Science Foundation for financial support.

REFERENCES

[1] Buchman, Gunther, Smith, Wang (eds.), Design and implementation of large spatial databases, Proc. 1st Symp. SSD '89, Santa Barbara; *Lecture Notes in CS 409*, Springer Verlag (1989).
[2] O. Gunther, Efficient structures for geometric data management, *Lecture Notes in CS 337*, Springer Verlag (1988).
[3] O. Gunther and H.-J. Schek, Advances in spatial data bases, Proc. 2nd Symp. SSD '91, Zurich; *Lecture Notes in CS 525*, Springer Verlag (1991).
[4] A. Hutflesz, H.-W. Six and P. Widmayer, The R-file: An efficient access structure for proximity queries, *Proc. 6th Intnl. Conf. on Data Engineering*, pp. 372–379 (1990).
[5] V. H. Nguyen, Implementation and performance of guard files, in preparation.
[6] J. Nievergelt and K. H. Hinrichs, Storage and access structures for geometric data bases. *Proc. Kyoto* 85 *Intern. Conf. on Foundations of Data Structures* (eds. Ghosh *et al.*), pp. 441–455, Plenum Press, NY (1987).
[7] J. Nievergelt, 7 ± 2 criteria for assessing and comparing spatial data structures, pp. 3–27 in [1].
[8] M. Overmars, personal communication.
[9] H. Samet, *The design and analysis of spatial data structures*, and *Applications of spatial data structures*, Addison-Wesley (1989).
[10] P. Schorn, Implementing the XYZ GeoBench: A programming environment for geometric algorithms, in H. Bieri and H. Noltemeier (eds.), Computational geometry: methods, algorithms and applications. *Proc. CG'91, International Workshop on Computational Geometry*, Bern, March 1991, Springer LNCS, pp. 187–202 (1991).
[11] P. Skrotzky, Implementing guard algorithms on the XYZ GeoBench, Diploma thesis, ETH Zurich (September 1992).
[12] P. Widmayer, Datenstrukturen für Geodatenbanken, pp. 317–361 in G. Vossen, K. U. Witt (eds.), *Entwicklungstendenzen bei Datenbanksystemen*, Oldenbourg (1991).